ISI Reprint Series
ISI/RS-94-410
June 1994

The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems

B. Clifford Neuman and Santosh Rao

ISI/RS-94-410
June 1994

University of Southern California
Information Science Institute
4676 Admiralty Way, Marina del Rey, CA 90292-6695
310-822-1511

This research was supported in part by the Advanced Research Projects Agency under NASA Cooperative Agreement NCC-2-539. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or NASA.

CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 6(4), 339–355 (JUNE 1994)

The Prospero Resource Manager: A scalable framework for processor allocation in distributed systems*

B. C. NEUMAN AND S. RAO
Information Sciences Institute
University of Southern California

SUMMARY

Existing techniques for allocating processors in parallel and distributed systems are not suitable for use in large distributed systems. In such systems, dedicated multiprocessors should exist as an integral component of the distributed system, and idle processors should be available to applications that need them. The Prospero Resource Manager (PRM) is a scalable resource allocation system that supports the allocation of processing resources in large networks and on multiprocessor systems.

PRM employs three types of managers—the job manager, the system manager and the node manager—to manage resources in a distributed system. Multiple independent instances of each type of manager exist, reducing bottlenecks. When making scheduling decisions each manager utilizes information most closely associated with the entities for which it is responsible.

1. INTRODUCTION

Multiprocessor systems are expensive and should be utilized to the fullest extent possible. When excess processing cycles are available, those cycles should be available to others. To promote such sharing, multiprocessor systems must exist in the broader context of distributed systems. Conventional techniques for managing resources in parallel systems perform poorly in large distributed systems. This paper describes the Prospero Resource Manager (PRM), a tool for managing processing resources in distributed parallel systems. PRM manages resources at two levels: allocating system resources to jobs as needed (a job is a collection of tasks working together), and separately managing the resources assigned to each job.

Developed for use by the Prospero operating system, under development at the University of Southern California's Information Sciences Institute, PRM presents a uniform and scalable model for scheduling tasks in parallel and distributed systems. PRM provides the mechanisms through which nodes on multiprocessors can be allocated to jobs running within an extremely large distributed system.

Figure 1 represents the user's view of a parallel application executed by PRM. The user sees the application as if it were sequential. Behind the scenes, PRM selects the processors on which the application will run, starts the application, supports communication between the tasks that make up the application, and directs input and output to and from the terminal and files on the user's workstation.

*Based on 'Resource Management for Distributed Parallel Systems' by B. C. Neuman and S. Rao which appeared in 2nd International Symposium on High-Performance Distributed Computing (HPDC-2), Spokane, WA, USA, July 1993: pp. 316–323.

CCC 1040-3108/94/040339-17

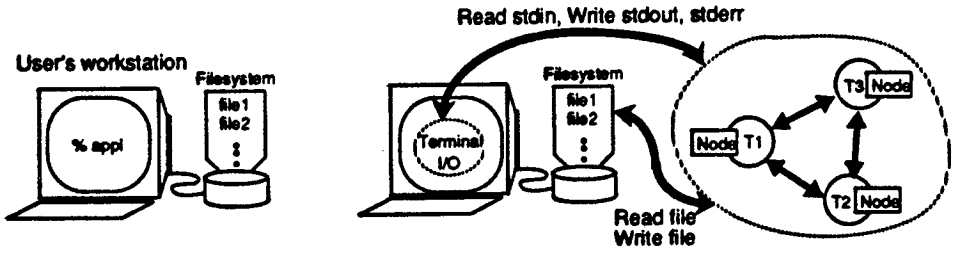

Figure 1. Users' view of job execution under PRM: (a) user invokes the application program on his workstation; (b) a set of tasks are created. Tasks execute the program, communicate with each other and perform terminal and file I/O.

To be useful in an environment such as ours, resource management techniques must scale numerically, geographically and administratively. The common approach of using a single resource manager to manage all resources in a large system is not practical. As the system grows, a single resource manager becomes a bottleneck. Even within large local multiprocessor systems the number of resources to be managed can adversely affect performance. As a distributed system scales geographically and administratively, additional problems arise[1].

PRM addresses these problems by using multiple resource managers, each controlling a subset of the resources in the system, independent of other managers of the same type. The functions of resource management are distributed across three types of managers: system managers, job managers, and node managers. The complexity of these management roles is reduced because each is designed to utilize information at an appropriate level of abstraction.

While the development of PRM was motivated by the desire to support parallel computing across organizations in a distributed system, the same techniques can improve the scalability of scheduling mechanisms within independent tightly coupled multiprocessor systems. The abstractions provided also naturally suggest extensions that support fault-tolerant and real-time applications and debugging and performance tuning for parallel programs.

Throughout this paper we use the term *node* to denote a processing element in a multiprocessor system, or a workstation or other computer whose resources are made available for running jobs. A *job* consists of a set of communicating tasks, running on the nodes allocated to the job. A *task* consists of one or more threads of control through an application and the address space in which they run.

2. CONTEMPORARY APPROACHES

Figure 2 shows the phases of execution of an application in a distributed environment. In the first step (1), the application is compiled and installed and information about resource requirements and available resources are specified by the user or programmer. This information is used in (2) to select and allocate nodes on which the program will run. The tasks are mapped to the allocated nodes in (3) and the executable modules (the tasks) are loaded onto the appropriate nodes in (4). The execution of the program (5) depends on run-time communication libraries (also at 4) which in turn use information about the mapping of tasks to nodes (3).

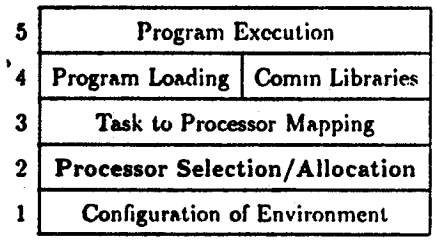

Figure 2. Application execution in distributed environments

Locus[2], NEST[3], Sprite[4], and V[5] support processor allocation, and remote program loading and execution (2,4,5) to harness the computing power of lightly loaded nodes. They primarily support sequential applications where task-to-task communication is not required. A critical issue for processor allocation in these systems is the maintenance of the database of available nodes. In Locus, the target node for remote execution is selected from a list of nodes maintained in the environment of the initiating process. This approach is inflexible because the list of nodes is fixed and dynamic load variations on the nodes are not considered.

In NEST[3], idle machines advertise their availability, providing a dynamically changing set of available nodes; each user's workstation maintains the list of servers available for remote execution. Locus and NEST both require the application to maintain information about every possible target node, limiting the size of the pool from which nodes can be drawn. Additionally, resource allocation decisions in these systems are made locally by the application without the benefit of a high-level view across jobs. This causes problems when applications run simultaneously.

Sprite[4] uses a shared file as a centralized database to track available nodes. Clients select idle nodes from this file, marking the entry to flag its use. While this approach appears simple, it requires a solution to problems related to shared writable files, including locking, synchronization and consistency. Fault-tolerance is also poor, since failure of the file server on which the shared file resides disables the allocation mechanism completely. This approach does not scale beyond a few tens of nodes.

Theimer and Lantz experimented with two approaches for processor allocation in V[5]. In a centralized approach a central server selects the least loaded node from a pool of free nodes and allocates it. Nodes proclaim their availability based on the relationship of the local load to a cutoff broadcast periodically by the server. This approach has limited scalability and poor fault tolerance since the central server is a critical resource. In the distributed approach a client multicasts a query to a group of candidate machines selecting the responder with the lowest load. This approach suffers from excessive network traffic and was found to perform worse than the central server approach.

The UCLA Benevolent Bandit Laboratory (BBL)[6] provides an environment for running parallel applications on a network of personal computers. Like the other systems discussed, BBL provides processor allocation, and remote program loading and execution (2–5), incorporating the notion of a user-process manager separate from a systemwide resource manager. While this is an important step towards scalable resource management techniques, a single resource manager will be unable to handle all allocation requests for a large system.

DQS[7] and Lsbatch[8] are load sharing systems which emphasize processor allocation (2) and are used mainly for distributing batch jobs across machines. Parallel jobs are supported but facilities for intertask communication (4) are minimal. Both packages are based on a central server that maintains load information and dispatches queued jobs. As with the V approach, their scalability is limited.

Parallel Virtual Machine (PVM)[9] and Net-Express[10] provide environments for parallel computing on a network of workstations. In the initial configuration phase, users specify a list of nodes on which they have started daemon processes. Based on this configuration, PVM and NetExpress map a job's tasks to nodes, load and execute the tasks, and support communication between tasks (3–5). There is no support for high-level resource allocation functions (2) that assign nodes to jobs with the goal of efficient system utilization. All nodes specified by the user are available to the job even though they may already be in use by other jobs.

Efficient management of a pool of processors becomes very important when the system scales to large numbers of nodes, spanning multiple sites. The emphasis of the Prospero Resource Manager is the allocation of nodes across and within jobs (2). The job manager eliminates the need for users to enumerate all hosts on which their applications might run, while the system manager efficiently manages the system's resources. While PRM also supports task mapping, program loading and execution (3–5), it is the allocation function (2) that distinguishes it from PVM and NetExpress. This function complements the features of PVM and NetExpress, and an integration of the PRM allocation methods with both packages would benefit all three. We have implemented a library that supports PVM applications on top of PRM.

3. SCALABLE RESOURCE MANAGEMENT

Users have difficulty dealing with extremely large systems because, although only a small subset of the available resources are needed, it is difficult to identify the resources that are of interest among the clutter of those that are not. Today's users are able to cope because only a tiny portion of the world's resources are available to them. Managing the world's resources is a daunting task, but the problem is simplified when it is reduced to managing only a subset of the resources.

We believe it should be possible to organize *virtual systems* in which resources of interest are readily accessible, and those of less interest are hidden from view. The organization of such systems should be based on the conceptual relationship between resources, and the mapping to physical locations should be hidden from the user. These concepts form the basis of the Virtual System Model, a new model for organizing large distributed systems[11].

To apply the concepts of the Virtual System Model to the allocation of resources in large systems, we have chosen to divide the functions of resource management across three types of managers: the system manager, the job manager and the node manager. The system manager controls a collection of physical resources, allocating them to jobs when requested. The job manager is responsible for requesting the resources needed by a job, and once allocated, assigning them to the individual tasks in the job. The node manager runs on each processor in the system, loading and executing tasks when authorized by the system manager and requested by the job manager. Each manager makes scheduling decisions at a different level of abstraction, some concerned with the high-level performance of the system, and others concentrating on particular jobs.

3.1. The system manager

The full set of resources that exist in a system will be managed by a set of system managers. For example, one or more system managers might manage the nodes in a parallel computer, or the resources local to a particular site. System managers allocate their resources across jobs as needed. We do not believe that it is possible to build a single system manager to manage all resources in a large system. As the system to be managed grows, a single system manager would become a bottleneck. To avoid this problem, our system supports multiple system managers, each responsible for a collection of resources. For example, one system manager can be responsible for the processors on a multiprocessor system and when necessary for performance or other reasons, multiple system managers may exist, each controlling a subset of the processors on the multiprocessor. An independent system manager might manage the resources available on one or more workstations.

The system manager is a hierarchical concept. Several sets of resources may be managed by different system managers, with a higher level system manager responsible for the entire collection. The control of resources could then be transferred from one system manager to another as directed by the higher level manager.

The system manager keeps track of the resources for which it is responsible, maintaining information about the characteristics of each resource, whether it is currently available, and if assigned, the job to which it is assigned. The system manager responds to status updates from node managers and resource requests from job managers. Status updates provide information needed to make allocation decisions including availability and load information. Resource requests identify the resources required by a job, their characteristics, as well as connectivity constraints, but only in well defined ways. It is possible to extend the system manager to accept messages from higher level managers (or other entities) adding or removing resources from its control.

When a resource request is received from a job manager, the system manager determines whether suitable resources are available as defined by the characteristics specified in the request. If so, the system manager assigns them to the job, notifies the node managers responsible for each resource that the resource has been assigned to a specific job manager, and informs the job manager of the resources that have been assigned. If the requested resources are not available the system manager can, at the job manager's option (and subject to the scheduling policy of the system manager), assign a subset of the requested resources and/or reserve the resources for assignment when they become available.

3.2. The job manager

Although multiple system managers are necessary for scalability, the application needs a single point of contact for requesting resources. In our system, this point of contact is the job manager. The job manager acts as an agent for the tasks in a job, providing a single entity from which the tasks will request resources. In this capacity the job manager provides the abstraction of a virtual system to a job, managing the resources that have been allocated to a job by the system managers responsible for each resource. Although it is possible for a job to have more than one job manager, in most cases only one exists.

The job manager is part of a job and is aware of the specific requirements and communication patterns of the tasks it manages. As such, the job manager is better able than the system manager to allocate resources to the individual tasks within a job. This is the same argument used in favor of user-level thread management on shared-memory multiprocessors[12]. In fact, we allow the job manager to be written by the application programmer if specific functionality is required, though we do not expect this to be a common practice.

We plan to eventually provide alternative job managers to support fault-tolerant and real-time applications. Such job managers would add additional requirements to the resources requested from system managers, and might assign individual tasks to multiple nodes. Similarly, we are developing a job manager that will collect information needed for debugging and performance tuning. The programmer would then be able to select job managers tailored to the needs of the application or the phase of program development; when an application is ready for production use, a different job manager could be substituted.

At the time a job is initiated, the job manager identifies the job's resource requirements. Using the Prospero Directory Service[13], if available, or a configuration file otherwise, it locates system managers with jurisdiction over suitable resources and sends allocation requests. If the system managers respond affirmatively, the job manager allocates the resources to the tasks in the job, contacting the node manager for each resource to initiate the loading of programs onto the appropriate processors. If the system manager refuses the allocation request, the job manager will try to identify alternate resources from other system managers. If necessary, the job manager will additionally create tasks to handle I/O to the terminal or to files on the local system.

Once the job has been initiated on the assigned nodes, the job manager monitors the execution of the program. During program execution the job manager responds to requests from the job's tasks for additional resources, reallocating them from other tasks or requesting additional resources from suitable system managers. In this phase, the job manager also maintains information about the mapping of logical task identifiers to node identifiers for use by the communication library.

3.3. The node manager

The third component of our resource management suite is the node manager. A node manager runs on each processor in the system, eventually as part of the kernel but in the current implementation as a user-level process. The node manager accepts messages from the system manager identifying the job managers that will load and execute programs. When requested by an authorized job manager, the node manager loads and executes a program. The node manager notifies the job manager about events such as the termination and failure of tasks. The node manager also keeps the system manager informed about the availability of the node for assignment. The node manager caches information needed to direct messages for other tasks to the node on which the task runs.

3.4. Application invocation

Each program that executes under PRM has associated with it information about the virtual system on which it will run. This information is stored either in a configuration

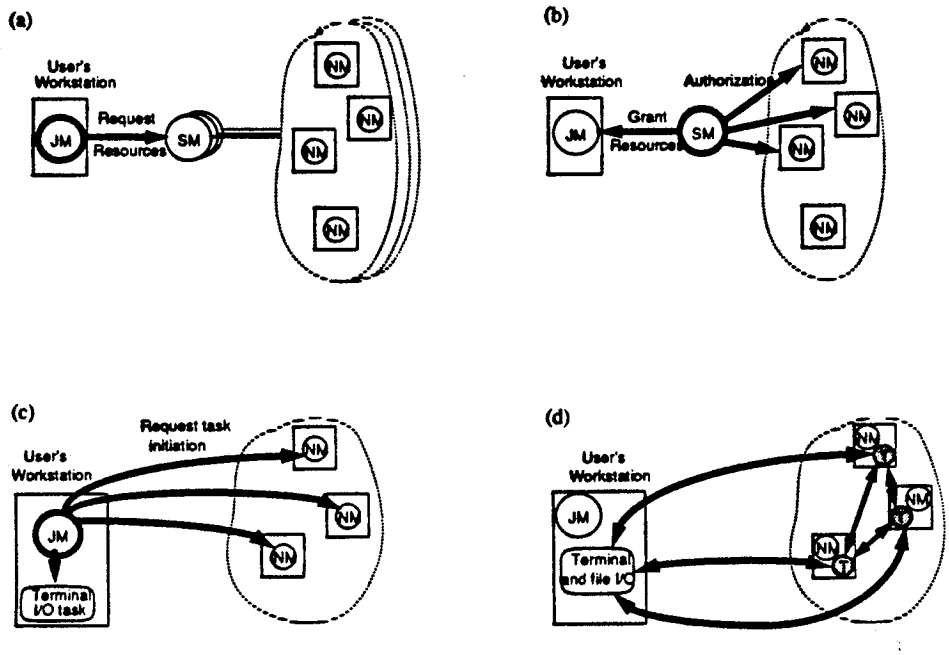

Figure 3. Job execution by the Prospero Resource Manager: (a) job manager identifies the resource requirements of job, requests them from one or more system managers; (b) system manager allocates resources to job manager, authorizes job manager to use node managers; (c) job manager requests node managers to fork tasks on the nodes; starts terminal and file I/O locally; (d) tasks communicate with each other and perform terminal and file I/O through the I/O tasks.

file or as attributes of the program in the Prospero Directory Service[13]. When a program is invoked, a new job manager is created and the job manager finds a suitable processor or set of processors by contacting system managers identified by the virtual system associated with the program.

Figure 3 shows the steps involved in running an application using PRM. Our goal is for users to invoke programs as if they were local to a workstation. When the program is invoked, a job manager is automatically started on the workstation[1]. The job manager determines the resource requirements of the job and sends requests to one or more system managers. If the requested resources are available the system manager informs the node manager responsible for each resource that the resource has been assigned to a particular job manager and it returns a list of the assigned resources to the job manager. The job manager further allocates the assigned resources to the job's tasks then contacts the node manager for each resource to invoke the application. Upon receipt of a request from the authorized job manager, each node manager loads the application task.

[1] In some cases though, it might migrate to another node.

During job execution, the job manager responds to requests from the job's tasks for additional resources (additional processors, for example) and to pre-emption and migration requests from the system managers responsible for the resources in use and, if necessary, attempts to obtain additional resources from other system managers. The job manager acts as an agent for the user, hiding the details of parallel execution; the user's environment and shell are maintained on the workstation and the job manager decides where each command is to execute, hiding the details from the user for whom local sequential execution and parallel execution appear identical.

3.5. Discussion

By separating the job and system manager, the system manager becomes much simpler. The system manager is concerned only with the allocation of resources between jobs, eliminating application specific functionality. The job manager is part of a job, and has more information about the requirements and communication patterns for the tasks it manages. Thus, the job manager is in a better position to allocate resources to tasks once resources have been allocated to the job. Because the job manager is part of the job, it can be customized or even rewritten if application specific functionality is required.

4. IMPLEMENTATION

The current implementation of the Prospero Resource Manager runs on a collection of SPARCstations, Sun-3 and HP9000/700 workstations, connected by local or wide-area networks. Heterogeneous execution environments are supported—a system manager may manage nodes of more than one processor type. In the common case, there is one system manager for every site. For example, our set-up consists of one system manager responsible for a set of SPARCstations on USC's main campus, another managing a collection of SPARCstations, Sun-3, and HP700 workstations at ISI, 15 miles away from the main campus, while a third manages a set of HP700 workstations at MIT, on the other side of the country.

A job manager can acquire nodes of more than one processor type, or nodes from more than one system manager. If necessary, the user may place constraints on the type and location of nodes through job configuration options. Normally, tasks within a job may execute on different processor types[2], and the set of nodes executing a job need not share a common filesystem. In the latter case, PRM handles program loading and I/O to shared files. As was shown in Figure 1, the user's view of job execution is essentially that of a locally executing sequential program.

While PRM is mainly intended for parallel jobs, remote execution of sequential applications is also supported. Parallel programs can be coded in C and must be linked with communication libraries described in Section 4.2, but no modifications are required for sequential programs. In the latter case, input and output will be redirected by the node manager to the terminal and file I/O tasks.

Multiple users may share a common PRM environment. Once set up, system and node managers run as server processes. The system manager maintains the *availability status* of each node, and allocates available nodes to jobs when requests are received. Depending

[2] It is assumed that program binaries are available for each processor type.

on options specified at set up, a node manager may make its processor unconditionally available for running jobs, or available only within specified time windows, or when no user is logged onto the workstation (or a combination of the latter two options). Node managers continually monitor node status and notify their system manager of any changes in node availability.

The PRM package may be used standalone or configured to use the Prospero Directory Service (PDS)[13]. If configured standalone, the user specifies job configuration information in a *job-description file*. Such information may include the minimum number of nodes required to run the job, constraints on the node types, if any, path names and binary types of executable files, the number and location of I/O tasks and the host names of one or more system managers that can potentially satisfy the resource requirements. If installed with the PDS, configuration information is organized as nodes in the file system hierarchy. A PRM program name corresponds to the name of a directory, and under it are links to the executable files required to run the application. Also specified as attributes of this directory are resources that are to be used when an application is run. At run time, the PDS maps resource names to the controlling system manager.

A user initiates a job by invoking a job manager process on the workstation, specifying a configuration file from which resource requirements of the job are to be read. In a future release, static resource requirements will be generated by a compiler (possibly with some help from the programmer) and stored as file attributes using the PDS. At runtime, the user may override static specifications and specify runtime requirements as command line arguments to the job manager.

4.1. Program loading and I/O

When one or more of the nodes assigned to a job do not share a common filesystem with the node on which the task binaries reside, explicit loading of files is supported by PRM. For this purpose, the job manager starts up a file I/O task, which co-operates with the node managers in transfering the executable files to the nodes' local filesystems.

The file I/O task also handles access to files on the user's local system. The job manager schedules these I/O tasks on nodes with local I/O devices. For example, an I/O task may run on a file server, performing file I/O on behalf of the tasks. At any given time, a task has exclusive read and write access to a shared file.

Terminal I/O is handled by another special task created by the job manager if needed. The terminal I/O task supports interactive execution. It is analogous to the host task in some other environments. Users can customize this task for job initialization functions, such as prompting the user for interactive input and distributing this input to the appropriate tasks. The I/O functions included in the communications library use the Prospero Data Access Protocol (PDAP)[14] for data transfer.

4.2. Communication libraries

To support parallel applications, several communications libraries are available with the PRM package. One library provides routines for sending, receiving and broadcasting tagged messages. Its application programming interface (API) is similar to that provided on multicomputers such as the Touchstone Delta[15]. Another API provides most of the commonly used routines available to Connection Machine (CM-5) programmers. This

interface implements CMMD library routines[16] through a set of macros and functions that make equivalent calls to our own library. The terminal I/O task takes on the functions of host program on the CM-5. By linking with this library, programs written for the CM-5 run virtually unmodified in PRM's environment. Thus, PRM can serve as an application testbed for CM-5 programs. The *Ocean* program from the SPLASH benchmark suite from Stanford University[17] studies the role of eddies and boundary currents in influencing large-scale ocean movements by solving a set of partial-differential equations. We have ported a CM-5 version of *Ocean* to PRM and are using it as a test application to tune our communication libraries.

A third library exports an interface identical to that of PVM version 3.2.6[18]. PVM's routines for message passing, buffer manipulation, process control and data packing and unpacking are available, enabling most PVM applications to run in the PRM environment without modification, giving them the added benefit of PRM's automatic resource allocation mechanisms. We are currently implementing an interface for PVM's dynamic process group library.

In developing these libraries a layered approach has been used to facilitate easy integration of the user-level routines with a variety of low-level communication protocols. In the present implementation, the libraries use an Asynchronous Reliable Delivery Protocol (ARDP) to transmit and receive sequenced packets over the Internet using UDP. We are also implementing a version of the communication library layered on top of the Mach *port* mechanism[19]. This will enable the three types of managers to communicate via Mach IPC, which will in turn pave the way for server-based implementation of PRM on multicomputers running Mach.

These communication libraries provide location transparency at the application level, freeing the programmer from having to keep track of task-to-node mappings. Applications programs address messages using logical *task-identifiers* (*tids*), which are translated to an internet-address/port pair within the library. When a pair of tasks communicate for the first time, the node manager assists in this translation using a mapping table furnished by the job manager. Such translations are then cached in the local address space of the task to reduce address translation overhead for subsequent communication with the same task.

4.3. Job managers supporting program development

As stated in Section 3.2, job managers can be tailored to specific phases of program development. We are developing a job manager that supports debugging of parallel applications. Using a checkpoint-and-replay approach in which each task maintains a log of its communication activity and saves its state periodically, this debugger can restore programs to past global states and replay events in the same order as the original execution. By specifying a set of *rollback predicates* the programmer can replay individual tasks in isolation, subsets of tasks, or the entire parallel program. To ensure that rollback states are consistent with the original execution, algorithms based on the assignment of logical timestamps to events are used. These consistency checks are performed at replay time to minimize the intrusiveness of the monitoring activity at run time.

The *debugging job manager* provides an interactive front-end through which the programmer monitors and controls the execution of the application program. To actually manipulate message logs, perform consistency checks and other debugging functions on

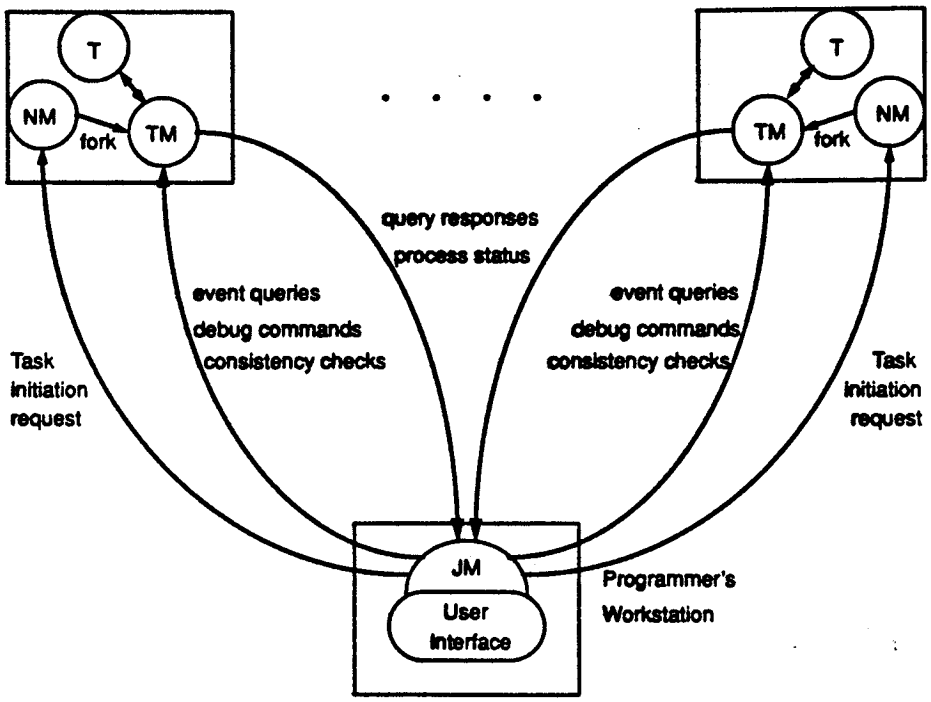

Figure 4. A job manager supporting debugging of PRM applications

a task, it is convenient to implement a set of back-end debugger processes known as *task monitors*. One task monitor exists for each task and is co-located with the task. Figure 4 shows the interaction of the various entities involved in debugging a PRM application. Checkpoints, consistency checks and replay in the context of PRM are discussed in greater detail in[20].

5. PERFORMANCE

Two types of experiments were conducted to measure the performance of PRM. The first experiment was intended to determine the communication latencies observed by application programs and compare the performance of our implementation of the PVM library over ARDP with that of PVM version 3.2.6 available from Oak Ridge National Laboratory[18]. The second experiment was designed to measure the the resource allocation performance of PRM. These experiments were conducted on a set of SPARC-10s connected by an Ethernet. To minimize the effects of cross traffic on the network and scheduling conflicts with competing processes on the workstations, the test programs were run when no other user processes were running on any of the workstations on this Ethernet. The workstations ran a modified version of SunOS 4.1.3 with an improved timer facility to increase the accuracy of measurement. For each run, elapsed times were measured using the `gettimeofday()` Unix system call. The figures in Tables 1–4 are in milliseconds and represent averages computed over several runs.

Table 3. Allocation time as a function of the number of nodes allocated

	Number of nodes			
	2	4	8	12
Delay	14.7	19.1	29.2	39.7

Table 4. Allocation time as a function of the number of system managers from which resources are requested. A total of eight nodes were allocated in each case

	Number of system managers			
	1	2	4	8
Delay	29.2	36.5	51.4	76.1

parallel jobs. In these experiments, the job manager was unaware of the resource configurations that the system managers controlled, and therefore had to query each one sequentially. By using the Prospero Directory Service, the job manager will be able to easily identify those system managers that can potentially satisfy its resource requirements and submit resource requests in parallel. The time spent in propagating authorization information can be reduced considerably by using asynchronous messages and improved broadcast mechanisms.

6. FUTURE DIRECTIONS

The current implementation of PRM demonstrates only a few of the benefits of our resource management model. The greatest benefit of the model is its flexibility. The prototype provides a framework within which we can try experimental solutions to interesting problems, and upon which interesting tools may be built.

Among our planned experiments are the use of interchangeable scheduling policies by the system manager. Because our model supports multiple system managers, multiple scheduling policies can be applied simultaneously to disjoint sets of resources. We also plan to explore options for hierarchical configuration and dynamic reconfiguration of the nodes for which a system manager is responsible. We will extend the job manager to make use of information such as task memory and I/O requirements, and intertask communication graphs to place constraints on the resources requested from system managers.

The components of a distributed system are usually owned by multiple users who may impose different constraints on the jobs that make use of their resources. To enable tasks to relinquish resources when reclaimed by owners, we are adding support for pre-emptive scheduling of tasks. Pre-emptive scheduling can also be applied when insufficient processing resources are available to tasks within a job. Extending this a step further, task migration mechanisms enable pre-empted tasks to acquire new resources and continue execution on a different node. We plan to implement task migration using the checkpointing mechanisms provided by Condor[21]. Checkpointing will also be useful for playback debugging described in[20].

The Prospero Resource Manager provides a basic framework for acquiring and managing resources in distributed environments. We plan to extend this framework to provide an integrated environment consisting of a set of tools that facilitate the development and execution of parallel and distributed applications. As described in Section 4.3 we are currently developing a job manager that supports application debugging. We will also develop special job managers for fault-tolerant and real-time applications.

I/O to files is still a limiting factor for many applications. This is especially troublesome when the processors on which an application runs are separated geographically from the disk on which a file is stored. The file I/O task plays an important role, supporting read and write operations to files on computers that do not export their file system. To increase the efficiency of file acccesses, we are extending the Prospero Data Access Protocol[14] to support file caching.

Finally, with the ability to run applications on multiprocessor systems across wide-area networks, security will become a critical problem. It is unlikely that sites would make their resources available to others if there are no methods for protection. Security mechanisms are needed to control access to remote nodes, to allow remote tasks to securely retrieve data that might be stored across a wide-area network, and to account for the use of processing resources. We plan to incorporate further security mechanisms into the software we develop, concentrating initially on authentication and authorization mechanisms to be applied when a request is received by the system and node managers.

7. CONCLUSIONS

As a distributed system grows, resource management becomes increasingly difficult. While it is easier to manage resources in different parts of the system separately, such an approach makes life difficult for the users and programmers who must interact with more than one entity to obtain the resources needed by an application. Unfortunately, when the centralized management techniques used in today's parallel systems are applied in large distributed systems, the resource manager becomes a bottleneck.

The Prospero Resource Manager combines the best of both approaches. Individual managers control small collections of resources in the system, independent of other managers of the same type. The system manager controls resources that are physically or administratively related. The job manager controls resources that are logically related, i.e. those resources needed by a particular job. When resources are needed, an application requests resources from its job manager, which in turn requests the resources from one or more suitable system managers. The resources obtained are then reallocated across the tasks in the job. In this capacity the job manager acts as an agent for the user, making the system appear to the user as a single virtual system.

The Prospero Resource Manager provides a scalable approach to allocating processors in distributed systems. It also provides a framework upon which alternative scheduling algorithms can be evaluated and development tools implemented. The flexibility of PRM's model for resource allocation was demonstrated in Section 6, which described some planned future directions for our work. The use of our model in future distributed systems will provide the true test of its benefits.

ACKNOWLEDGEMENTS

The initial idea of separate job and system managers for scheduling tasks on a multiprocessor system was proposed by Stockton Gaines and Dennis Hollingworth. Ed Lazowska, John Zahorjan and Hank Levy contributed to discussions about the Virtual System Model. Jean Yun integrated the Prospero Directory Service with the Prospero Resource Manager. Rafael Saavedra provided code for the CM-5 version of Ocean. Bradford Clark, Ali Erdem and Ronald Wood implemented the functions in the communication library as part of a class project at USC. Performance experiments were conducted in the Operating Systems Laboratory in USC's Computer Science Department. Elliot Yan implemented the improved timer and the IP delay facility in the SunOS kernel. Celeste Anderson, Steven Augart, Gennady Medvinsky, Rafael Saavedra and Stuart Stubblebine commented on drafts of this paper. This paper is a revised version of a paper[14] that originally appeared in the 2nd IEEE Symposium on High Performance Distributed Computing, Spokane WA, July 1993.

REFERENCES

1. B. Clifford Neuman, 'Scale in distributed systems', in *Readings in Distributed Computing Systems*, IEEE Computer Society Press, 1994.
2. G. Popek and B. Walker (Eds.), *The Locus Distributed System Architecture*, M.I.T. Press, Cambridge, Massachusetts, 1985.
3. R. Agrawal and A. K. Ezzat, 'Location independent remote execution in NEST', *IEEE Trans. on Softw. Eng.*, **13**(8), 905–912 (1987).
4. F. Douglis and J. Ousterhout, 'Transparent process migration for personal workstations', Technical Report UCB/CSD 89/540, Computer Science Division, University of California, Berkeley, CA 94720, November 1989.
5. M. A. Theimer and K. A. Lantz, 'Finding idle machines in a workstation-based distributed system', *IEEE Trans. on Softw. Eng.*, **15**(11), 1444–1458 (1989).
6. R. E. Felderman, E. M. Schooler and L. Kleinrock, 'The Benevolent Bandit Laboratory: A testbed for distributed algorithms', *IEEE J. Sel. Areas Commun.*, **7**(2), 303–311 (1989).
7. T. P. Green and J. Synder, 'DQS, a distributed queueing system', in *Workshop on Heterogeneous Network-Based Concurrent Computing*, Supercomputer Computations Research Institute, Florida State University, October 1991.
8. J. Wang, S. Zhou, K. Ahmed and W. Long, 'Lsbatch: A distributed load sharing batch system', Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto, Toronto, Canada, April 1993.
9. V. S. Sunderam, 'PVM: A framework for parallel distributed computing', *Concurrency: Pract. Exp.*, **2**(4), 315–339 (1990).
10. ParaSoft Corporation, Pasadena, CA 91107, *NetExpress 3.2 Introductory Guide*, 1992.
11. B. Clifford Neuman, *The Virtual System Model: A Scalable Approach to Organizing Large Systems*, PhD thesis, University of Washington, June 1992; available as Department of Computer Science and Engineering Technical Report 92-06-04.
12. Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska and Henry M. Levy. 'Scheduler activations: effective kernel support for the user-level management of parallelism', *ACM Trans. Comput. Syst.*, **10**(1), 53–79, (1992).
13. B. Clifford Neuman, 'The Prospero File System: A global file system based on the Virtual System Model', *Comput. Syst.*, **5**(4), (1992).
14. Steven Augart, B. Clifford Neuman and Santosh Rao. 'The Prospero data access protocol', in preparation.
15. Intel Supercomputer Systems Division, Beaverton, OR, *Touchstone Delta C System Calls Reference Manual*, April 1991.
16. Thinking Machines Corporation, Cambridge, MA, *'CMMD Reference Manual*, October 1991.

17. J. P. Singh and J. L. Hennessey, 'Finding and exploiting parallelism in an ocean simulation program: experience, results and implications', *J. Parallel Distrib. Comput.*, 15(1), (1992).
18. A. Giest, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, 'PVM 3 user's guide and reference manual,' Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.
19. David L. Black, David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan and David Bohman. 'Microkernel operating system architecture and Mach', in *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, April 1992, pp. 11–30,.
20. Santosh S. Rao and B. Clifford Neuman, 'A replay based debugger for distributed parallel environments', submitted for publication.
21. M. Litzkow and M. Solomon, 'Supporting checkpointing and process migration outside the Unix kernel', in *Usenix Conference Proceedings*, Winter 1992, pp. 283–290.
22. B. Clifford Neuman and Santosh Rao, 'Resource management for distributed parallel systems', in *Proc. 2nd International Symposium on High Performance Distributed Computing*, July 1993, pp. 316–323.

This research was supported in part by the Advanced Research Projects Agency under NASA Cooperative Agreement NCC-2-539. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or NASA. Figures and descriptions in this paper were provided by the authors and are used with permission. To obtain the software described in this paper, send mail to info-prospero@isi.edu asking for information about the Prospero Resource Manager. The authors may be reached at USC/ISI, 4676 Admiralty Way, Marina del Rey, CA 90292-6695, USA. Telephone +1 (310) 822-1511, email bcn@isi.edu, srao@isi.edu.